

Essential Skills for the Agile Developer

Chapter Excerpt: Avoid Over- and Under-Design

Alan Shalloway, Scott Bain, Amir Kolsky, Ken Pugh

Developers tend to take one of two approaches to programming. Many think they need to plan ahead to ensure that their system can handle new requirements that come their way. Unfortunately, this planning ahead often involves adding code to handle situations that never come up. The end result is code that is more complex than it needs to be and therefore harder to change—the exact situation they were trying to avoid. The alternative, of course, seems equally bad. That is, they just jump in, code with no forethought, and hope for the best. But this hacking also typically results in code that is hard to modify. What are we supposed to do that doesn't cause extra complexity but leaves our code easy to change? The middle ground can be summed up by something Ward Cunningham said at a user group: "Take as much time as you need to make your code quality as high as it can be, but don't spend a second adding functionality that you don't need now!" In other words, write high-quality code, but don't write extra code.

This chapter is admittedly more of a new mantra than it is a detailed description of a technique to implement. This chapter takes advantage of what we learned in Chapter 5, Encapsulate That!, and sets the groundwork for Chapter 11, Refactor to the Open-Closed.

A Mantra for Development

We believe developers should have a particular attitude when writing code. There are actually several we've come up with over time—all being somewhat consistent with each other but saying things a different way. The following are the ones we've held to date:

- Avoid over- and under-design.
- Minimize complexity and rework.
- Never make your code worse (the Hippocratic Oath of coding).
- Only degrade your code intentionally.
- Keep your code easy to change, robust, and safe to change.

Before we can discuss these mantras, we need to be clear what we mean by quality code. Appendix B, Code Qualities, provides a thorough explanation of the specific qualities referred to in this chapter. We'll give a brief summary of code quality here, but interested readers may want to read the more extensive narrative in the appendix.

The Pathologies of Code Qualities

It's often easier to see code qualities by discussing examples of when the qualities aren't present. Let's look at five common code qualities: cohesion, coupling, redundancy, readability, and encapsulation.

- **Cohesion.** Strongly cohesive classes are classes whose functions are all related to each other. Strongly cohesive methods are methods that do only one thing. The pathology of weak cohesion is classes or methods that do unrelated things. We've heard very weakly cohesive classes called "god objects" presumably because they are somewhat omniscient in that everything takes place in them.¹
- **Proper coupling.** Having well-defined relationships between objects makes them easier to understand and likely to inadvertently cause problems when changing code. The pathology of improper coupling is the occurrence of side effects—that is, unexpected errors due to making changes elsewhere.
- **No redundancy.** No redundancy is difficult to achieve. The more redundancy you have, the more time it will take to make changes. As we discussed in Chapter 4, Shalloway's Law and Shalloway's Principle, no redundancy is virtually impossible to achieve—but at least you want to make it so you don't have to find the duplication. Essentially, the pathology of redundancy is that when you make a change in one place, you have to make a change in another place.
- **Readability.** Readable code means you can understand what has been written. It requires intention-revealing names and is best achieved by using Programming by Intention (see Chapter 1, Programming by Intention). Unreadable code, of course, is code you can't understand when you read it. Poor names, tight coupling, and big methods/classes contribute greatly to the unreadability of code.
- **Encapsulation.** Encapsulation is more than mere data hiding. The type of an object is one of the most important things to hide. Design patterns are really about hiding: object type, cardinality, which function is being used, order, optional behavior, construction, and more. The pathology of encapsulation is when you must know how the code you are using is implanted in order to use it properly. This often means you know the implementation type of the object being used or know something about cardinality, order, and so on.

Avoid Over- and Under-Design

This essentially means you should put in the correct amount of design. Overdesign is putting in things that add complexity to the code that may or may not be needed. Note that the key word here is "complexity." We're not as worried about the time you take as much as we are about how you leave the state of the code. If the work you've done does not raise the complexity of the code you have, then no worries. In other words, putting in an interface where one may or may not be needed is not necessarily a bad thing if everyone understands interfaces. Interfaces aren't really complexity-adders in our mind. They are a holder for an idea. However, putting in a complex parameter list (or using a value object to hold a parameter, say, when one isn't needed) would be raising complexity.

Under-design is actually a euphemism for "poor code quality." We view under-design as having taken place when high coupling or weak cohesion is present. Typically, proper encapsulation is

also not present. So, avoiding overdesign means make your code changeable, but don't add things you don't need now. If you need them later, the changeability of the code will enable you to do that with less, if any, extra cost. Avoiding under-design mostly means making sure your code is changeable.

Minimize Complexity and Rework

Many people only partly understand the true nature of refactoring. Martin Fowler, in his excellent *Refactoring: Improving the Design of Existing Code*,² describes refactoring in the following way.

- *Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written.*

In the book, Fowler talks about refactoring as a method of cleaning up messy/poor code. However, there is another side to refactoring that Fowler doesn't talk about. This is refactoring code that is of high quality, when it comes to the code qualities we've been talking about, but that no longer has sufficient design because of new requirements. In other words, the book talks about how to clean up poorly written code (a good thing to know) but mostly ignores how to refactor good code that now must be changed to accommodate new requirements.³

We strongly suggest that refactoring good code when new requirements come so that the code is better able to accommodate the changes is a way to minimize complexity because you are deferring adding complexity until it is needed, but your code quality is high so there is no rework. We would contend that delaying extensions to code is not rework but a kind of just-in-time design. We'll talk explicitly about how to do this in Chapter 11, Refactor to the Open-Closed.

Never Make Your Code Worse/Only Degrade Your Code Intentionally

Existing code degrades one bit at a time (no pun intended). We suggest that team members do their best to not take shortcuts that makes their code worse. Sometimes this is difficult, however. It may be that legacy code makes it very difficult to add functionality properly without harming your code. To be realistic, we restate "Never make your code worse" to "Only degrade your code intentionally." Although this may sound funny, the alternative would be to make your code worse *unintentionally*.

One way to only degrade your code intentionally is to ensure you consider alternatives. One way to do this is to make a teamwide agreement that if developers can't figure out how to make a change without degrading code, they will tell another team member of the change they are thinking of making before they make the change. Note that we are not requiring getting

permission or even getting a better result. We're just suggesting you tell someone. This forces you to at least reflect a little. Our experience has shown us that a person will stop just short of a good solution because he or she is willing to do the first thing that comes to mind. Our approach forces people to think about things a bit more (sometimes a lot more because they don't want to admit to coworkers that they don't have good solutions).

Keep Your Code Easy to Change, Robust, and Safe to Change

Code should not be viscous. That means the effort to make changes should not be excessive. Viscosity can be avoided by having easy-to-understand, nonredundant code. Code should also not be brittle. That is, changes in one place should not break code in other places. This requires loosely coupled code, following Shalloway's principle (see Chapter 4, Shalloway's Law and Shalloway's Principle) and proper encapsulation. It is not sufficient to follow these two mantras alone, however. Although doing so may make it easy to change your code with less likelihood of breaking it, there are no guarantees. The only way to be assured that you can safely change your code is to have a full set of automated acceptance tests available.

A Strategy for Writing Modifiable Code in a Non-Object-Oriented or Legacy System

Many of the approaches we've discussed here are often met with this attitude: "That's a great idea, but I can't do it where I work because I'm using C." A variant of this is "That's a great idea, but I can't do it where I work because there is so much monolithic legacy code that I can't take advantage of object-oriented methods." There are other variants as well, but you get the idea. Although it is true that your existing software and the languages you are using provide certain constraints on what you can do, there are certain approaches you can *always* take. One of these is to consider the separation of concerns in a different way.

The idea is to separate the code that is particular to the application from the code that defines the application's architecture (or even system architecture).

One can think of a program as essentially an overall flow detailing the steps to be undertaken. For example, a sales-order system can have a variety of actions needed to work:

- Select customer.
- Get customer information.
- Select products to be sold.
- Get prices.
- Apply appropriate discounts.
- Total cost of sales order.
- Specify shipping.

Object orientation attempts to simplify this by creating objects that group responsibilities for the different implementing steps. These objects collaborate with each other and avoid coupling by having well-defined interfaces that hide their implementations. Unfortunately, if you can't (properly) use an object-oriented language, how can you get at least some of the value that comes from separating concerns? One way is to have each method in your code deal with only one of the following:

- The system architecture
- The application architecture
- The implementation of a step

For example, let's say you are writing embedded software that takes its input from a special bus in the form of string from which it extracts required parameters via a specialized method. Applications like this often take the following approach:

```
public function someAction () {
    string inputString;
inputString= getInputFromBus();
    if (getParameter(inputString, PARAM1)> SOMEVALUE) {
        // bunches of code
    } else {
        if (getParameter(inputString, PARAM2)< SOMEOTHERVALUE) {
            // more bunches of code
            // ...
        } else {
            // even more bunches of code
            // ...
        }
    }
}
```

The problem with this is lack of cohesion. As you try to figure out what the code does, you are also confronted with detailed specifics about how the information is obtained. Although this might be clear to the person who first wrote this, this will be difficult to change in the future (not counting the confusion that happens now). This gets much worse if one never makes the distinction between the system one is embedded in (which is determining the input method) and the logic inside the routine. For example, consider what happens when a different method of getting the string is used as well as a different method of extracting the information. In this case, the parameters are returned in an array:

```
public function someAction () {
    string inputString;
    int values[MAX_VALUES];

    if (COMMUNICATION_TYPE== TYPE1) {
        inputString= getInputFromBus();
    } else {
        values= getValues();
    }

    if ( (COMMUNICATION_TYPE== TYPE1 ?
```

```

    getParameter( inputString, PARAM1) :
    values[PARAMETER1]) > SOMEVALUE) {
    // bunches of code
} else {
    if ( COMMUNICATIONS_TYPE== TYPE1 ?
        getParameter( inputString, PARAM2) :
        values[PARAMETER2])
        < SOMEOTHERVALUE) {
        // more bunches of code
        // ...
    } else {
        // even more bunches of code
        // ...
    }
}
}
}

```

Pretty confusing? Well, have no fears, it'll only get worse. If, instead, we separated the "getting of the values" from the "using of the values," things would be much clearer.

```

public function someAction () {
    string inputString;
    int values[MAX_VALUES];

    int value1;
    int value2;

    if (COMMUNICATION_TYPE== TYPE1) {
        inputString= getInputFromBus();
    } else {
        values= getValues();
    }

    value1= (COMMUNICATION_TYPE== TYPE1 ?
        getParameter( inputString, PARAM1) :
        values[PARAMETER1]);
    value2= ( COMMUNICATIONS_TYPE== TYPE1 ?
        getParameter( inputString, PARAM2) :
        values[PARAMETER2]);

    someAction2( value1, value2);
}

public function someAction2 (int value1, int value2) {

    if ( value1 > SOMEVALUE) {
        // bunches of code
    } else {
        if ( value2 < SOMEOTHERVALUE) {
            // more bunches of code
            // ...
        } else {
            // even more bunches of code
            // ...
        }
    }
}
}

```

}

You must remember that complexity is usually the result of an increase in the communication between the concepts involved, not the concepts themselves. Therefore, complexity can be lowered by separating different aspects of the code. This does not require object orientation. It simply requires putting things in different methods.

Summary

Developers must always be aware of doing too much or too little. When you anticipate what is needed and put in functionality to handle it, you are very likely to be adding complexity that may not be needed. If you don't pay attention to your code quality, however, you are setting yourself up for rework and problems later. Code quality is a guide. Design patterns can help you maintain it because they give you examples of how others have solved the problem in the past in similar situations.